3-Level Behavioural Models for Semantic Web Services

Barry Norton, Carlos Pedrinaci, Laurent Henocque, Mathias Kleiner

3-Level Behavioural Models for Semantic Web Services

Barry Norton[1], Carlos Pedrinaci[1], Laurent Henocque[2], and Mathias Kleiner[2]

[1] Knowledge Media Institute, Centre for Research in Computing,
Open University, Milton Keynes, UK
{b.j.norton, c.pedrinaci}@open.ac.uk
[2] {Laboratoire des Sciences de l'Information et des Systèmes, Marseille, France
{laurent.henocque, mathias.kleiner}@lsis.org

Abstract. There are two types of behavioural model in the WSMO semantic description of services: an orchestration and a choreography, together called the interface. While an orchestration defines a service's behaviour as a composition of existing parts, a choreography is intended to document the conversation of messages exchanged with its client. In this paper we present a three-level model for behavioural descriptions, and how UML Activity Diagrams and the Cashew workflow model fit into this, building on existing work on the use of Abstract State Machines to define behaviour in WSMO.

1 Introduction

The Web Service Modeling Ontology (WSMO) provides for the semantic description of web services with the aim of automating the tasks of discovery, selection, composition, mediation, execution and monitoring [13]. It is claimed that there are two separate characteristics of web services that must be captured to form semantic web services (SWSs), enabling this: the *functional* and the *behavioural*. WSMO is notable for also describing the *goal*s of users ontologically, *i.e.* rather than simply representing requirements to the discovery task, the client's requirements are made a first-class element and used through the entire process, notably in mediation and execution; for this reason goals also have behavioural, as well as a functional, characteristics.

In this paper we concentrate of the behavioural descriptions of SWSs. Two types of behavioural description are associated with SWSs in WSMO: *choreography* and *orchestration*, together these form the *interface*. Choreography is attached to a web service description to describe the order which message exchange can be engaged in. More generally such descriptions may be called message exchange patterns, and other notions of choreography exist expressing message exchange patterns, but in WSMO this strictly means the communication between the service and its (single) client. A choreography is also associated with a goal in WSMO, not simply as a reversal of the message exchange roles: the messages themselves may be described in different ontologies, requiring *data mediation*, and the exchange pattern may be different, requiring *process mediation*.

An orchestration, on the other hand, considers more parties than the client, since it describes how the behaviour of a service is composed from serveral parts. In general this may involve both web services and goals, allowing the resolution of goals to services, via discovery, to be deferred until run-time. An orchestration must define control flow and data flow between its component parts. The WSMO working group has previously proposed that the same model of *ontologized abstract state machines* (ASMs), on which choreography has been based [26], can be re-applied to define orchestration [8]. There are a number of deficiencies in this approach however.

First, it is a WSMO principal that the connection between any two disimilar components be made using a first-class *mediator*, wherein mediation requirements may be specified. So, for instance, the connection between two ontologies separately conceptualising the same objects can be made using an (ontology-ontology) oo-mediator, and the connection between a goal and a web service that meets that goal can be made using a (web service-goal) wg-mediator. The ASM model for choreography does not have the possibility to include mediation in this way, and dataflow is implicit in the copying of data by transition rules.

Second, although ontologized ASMs are an elegant means for expressing behaviour over semantic data, they lack tool support and familiarity to most users, and are engaged by only a limited research community, whose results can be applied for the purposes of reasoning. The approach followed in the DIP project[3] is to build a stack of descriptions in which behavioural models can be expressed, as shown in Figure 1.

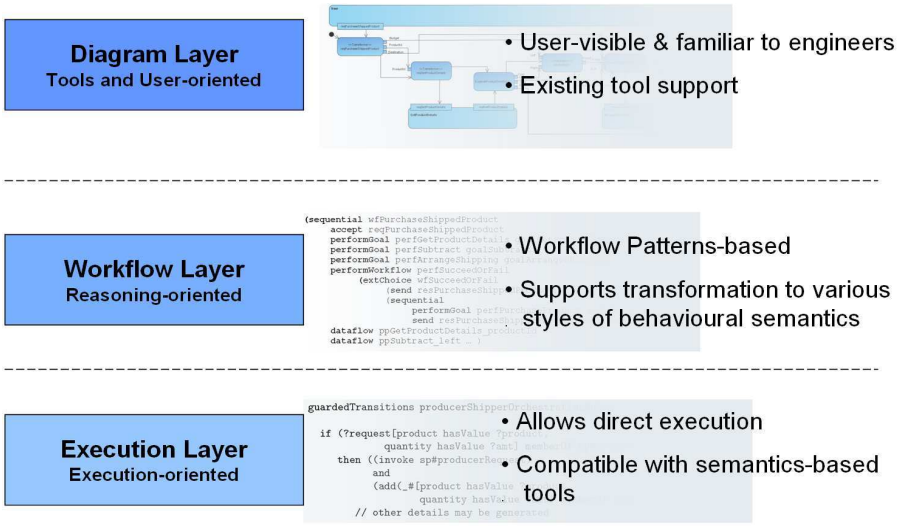


Fig. 1. 3-Layer Model

[3] http://dip.semanticweb.org

The presence of a middle-layer based on Workflow Patterns allows a standard approach to translation between various representations, see for instance [25], including, as well as execution, behavioural semantics in other forms allowing compositional and algebraic reasoning. Within DIP the three layers were fixed to be: UML2 Activity Diagrams; a WSMO-based evolution of the Cashew-S workflow ontology; and ontologized ASMs. The basis for each of these is described in the following section on Background work and then the formal meta-model developed described in Sections 3, 4 and 5 respectively, and Section 6 describes the relationship between the new models. Finally Section 8 concludes and discusses related work suggestion that this model is indeed general and a sound basis for re-application.

2 Background

2.1 WSMO and WSML

The fragment of the WSMO meta-model [9] we deal with is represented, as a UML class diagram, in Figure 2. The central concepts, duly shown centrally, are 'ontology', 'web service', 'mediator' and 'goal'.

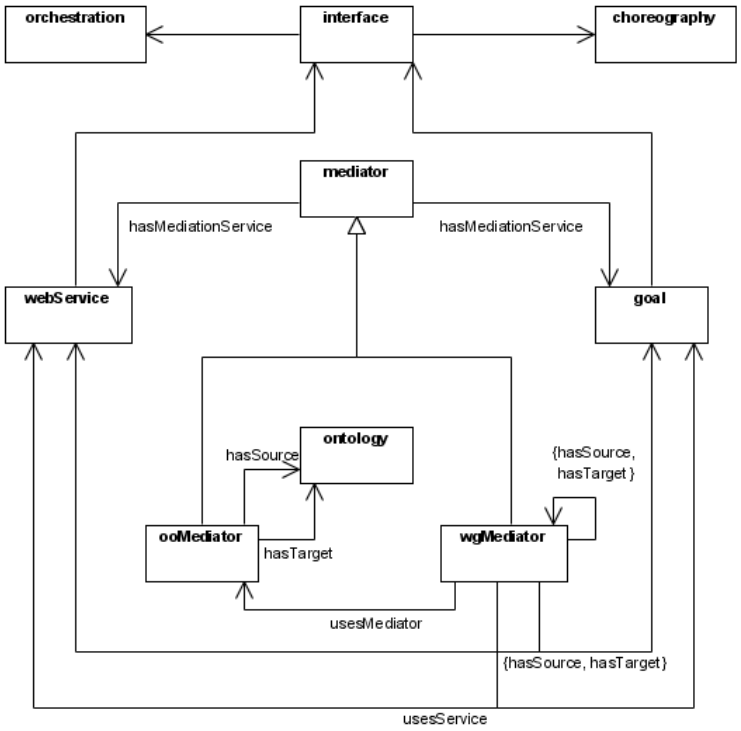


Fig. 2. (Partial) WSMO Meta-model

As described above, both goal and webService have functional and behavioural parts. The functional definition is in the *capability*, which we do not show, and the behavioural part is in the interface made up of choreography and orchestration descriptions. We do not consider the use of orchestrations for goals in this paper, but these may be understood to allow requirements for the composition of a service implementing the goal to be specified. We note that wg-mediator currently has two attributes to define the data mediation necessary to use a service to meet a given goal: an oo-mediator may map between the ontologies in which each is expressed; a goal or web service can be specified to provide data mediation that requires more complex computation.

The WSMO meta-model is expressed in the OMG's meta-object facility and then given a grammar to form the Web Services Modeling Language (WSML), in both a human-readable and an XML syntax, over which reasoning is defined.

2.2 OWL-S

Like WSMO, OWL-S [11] describes web services in both a functional and a behavioural form. The functional description is contained in the *service profile*, while the *process model* defines the behaviour. The OWL-S process model is an algebra of workflow forms, called processes, where the atomic processes are grounded to operations on web services. Although this has been called 'service composition', we have previously pointed out [19] that this is really a model of 'operation composition', *i.e.* a composite process is used to define a single operation, not a general service with multiple operations, and within that definition, the attachment to services of operations is not considered.

The gap between services and operations widens further when we consider statefulness of interaction in the service interface. When we allow that there is a 'protocol' governing the order of use of operations of a service in any given session, as defined by choreography in WSMO, it becomes clear that OWL-S neither defines nor respects this aspect of behaviour, defining only an orchestration over a simplified view of services.

There are two aspects of OWL-S that we consider useful in building on the current work in WSMO. First the hierarchical decomposition via control flow has proved particularly amenable to ontological description and to certain forms of automated composition, for instance via planning-based approaches. Secondly, the concept of an identified *performance* of an externally defined workflow, allowing the performance in different contexts of control and data flow, as a means of reuse, is something which we generalise on as a useful ontological construct.

2.3 Cashew-S

Earlier work defined a variation of the OWL-S process model, Cashew-S [20]. Although now superceeded by the Cashew model presented here, it is notable for two reasons. First it was the basis of a *compositional* semantics for OWL-S, an important property of behavioural semantics missed by previous work. Secondly, it suggested how a semantics based on process algebra could give rise

to an axiomatisation of behavioural equivalence, which would put behavioural reasoning within the scope of ontological reasoning.

2.4 Workflow Patterns

Workflow Patterns is a long-running project to formalise and account for the possible variations within workflow systems, and provide a common vocabulary to compare these [28]. It is telling that, when presented[4], one of the examples given of hidden differences between workflow systems in the early days was the distinction between, in workflow patterns terms, 'XOR' and 'Deferred Choice'. In general terms, when asked whether their systems supported a choice operator, vendors would answer 'yes'. On the other hand, given the vocabulary to ask whether systems supported choices resolved internally by the engine and choices resolved externally by the outcome of component tasks, the answer was too often only 'yes' one or the other, rather than both. In fact, this very example is another reason for the deficiency of OWL-S in the presence of choreography, since deferred choice is not supported by OWL-S. We shall show the extension to, and use of, this form of choice operator in the Cashew model, where control flow concepts are aligned with Workflow Patterns as a pre-existing 'shared conceptualisation', being exactly what ontologies are supposed to formalise.

2.5 UML

The UML is an ongoing effort by the Object Management Group [5], standardised by ISO [14], to provide a language for software engineering design via diagrams describing both the static and dynamic characteristics of software artifacts. Of particular relevance are UML2 activity diagrams (UML2AD) which, it has been suggested, are expressive enough to represent visually many of the workflow patterns [7, 29]. Most importantly, the workflow patterns can be represented modularly in UML2AD, which makes a significant difference with the situation known of UML1. The main change from UML1.5 to UML2 is that the activity diagram semantics have evolved from state machines to colored Petri nets.

Indeed the token flow semantics of UML2AD provide means to freely model and document the behavior and interface of semantic web services. The whole range of UML2AD constructs cannot however easily be used to those aims, because of doubts concerning the exact semantics of some of the constructs, which hence become unavailable for formal specification.

A central issue in UML2AD remains the potential to design activity diagrams that could block at execution because of starvation (a process waits forever for an event that will never occur) or interlocking (two processes mutually wait for each other to send a message). The UML2AD introduce "traverse to completion semantics" to reduce the risk of process starvation. According to those semantics, tokens remain in output pins until they may flow through edges for immediate

[4] Wil van der Aalst's 'Life After BPEL?', presented as keynote at WS-FM'05
[5] http://www.omg.org/technology/documents/formal/uml.htm

consumption by actions. This restriction does not however fully prevent the risk of starvation, because deadlocks may occur further down the flow or higher up because of loops, specially in the presence of duplicate output edges[6] and message input actions.

A fragment of the UML2AD that covers enough constructs to be used in the context of Semantic Web Composition has been isolated and specified in [4, 3] and refined in [23]. The metamodel for this subset will be presented and its abstract syntax formally specified using the Z notation in Section 5. The chosen AD subset has proved useful to implement constraint based automatic composition [4] in a context where web services document choreographies or orchestrations that cannot modularly be specified using boxed languages. Constraints in the meta-model warrant that automatically generated composite web services do not enter into trivial interlocking situations.

To illustrate the previous discussion on workflow patterns, UML2AD diagrams support internal choice using decision nodes on the one hand, and support external choice using interruptible activity regions on the other hand, as will be precisely shown in Section 6.

The technology used for composition amounts to finite model search for first order theories describes as constrained object models. The search process is often referred to as constraint based "configuration" in the literature. The choice of the UML2AD language is relevant in that context since the activity diagrams that represent the choreographies of semantic web services, as well as the activity diagrams that represent the orchestrations of composite web services occur as finite models of the UML2AD language. Finite model search can naturally be used for composition in that context, as the process may take as its input a partial description of the target composite activity involving the choreographies of the candidate participating web services, and complete this description by inserting new workflow constructs and actions.

3 Abstract State Machines

The current proposal for Choreography in WSMO [10], diagrammed in Figure 3, proposes that abstract state machines are ontologized so that the state is represented by the instances of a set of ontological concepts and relations (hereafter the document will refer only to concepts, leaving implicit that this also applies to relations) contained in a *state signature*. The state signature, as well as collecting these concepts, constrains the operations that may be made over the instances by transition rules. These operations are divided into tests and updates. Tests are recursive rules, *i.e.* decompose into other rules, and may inspect the instances and bind variables; update rules may add, delete or change the instances of a concept. The state signature attaches *modes* to concepts, as follows:

– concepts of IN mode can be tested on, but neither populated, nor have their instances updated or deleted;

[6] If two edges get out of the same pin, it means that a token may take either direction, according to the traverse to completion semantics.

- concepts of OUT mode can be populated, but not the tested on;
- concepts of SHARED mode can be tested on and populated, and the instances updated;
- concepts of STATIC mode can be tested on, but the instances can be neither updated, created or deleted;
- concepts of CONTROLLED mode can be tested on and have their instances created, deleted and updated.

Modes are also the way in which (syntactic) grounding is assigned to concepts to be communicated. IN, OUT and SHARED mode concepts can be given groundings, for instance via WSDL to messages, via interfaces and operations.

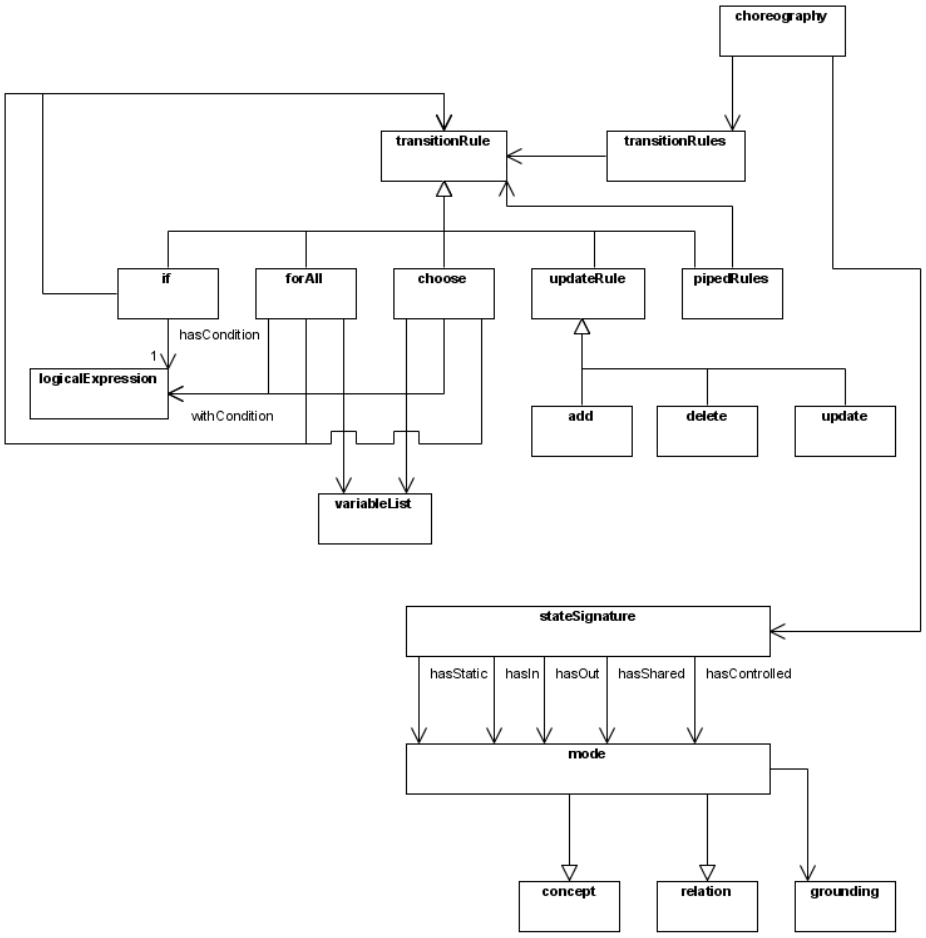


Fig. 3. WSMO Choreography Meta-model

It was previously proposed by the WSMO working group to introduce a set of explicit 'invoke' transition rules into the grammar for orchestration ASMs [8], so that web services, goals and mediators could be explicitly included. The proposal resulting from this work goes one step further and suggests that each such prototypical invocation, having a distinct control and data flow context, should be recognised as a first-class ontological element, named perform [21]. Thereafter a perform rule, which defines any one of the subtypes of perform shown in the additional meta-model shown in Figure 4, is added to the ASM grammar for orchestration. Finally, a new mediator, the pp-mediator, can be used to represent the dataflow between performances of all these elements, as well as asynchronous communications, in order to meet the WSMO principal mentioned above that all such links should use an explicit mediator to specify the necessary mediation.

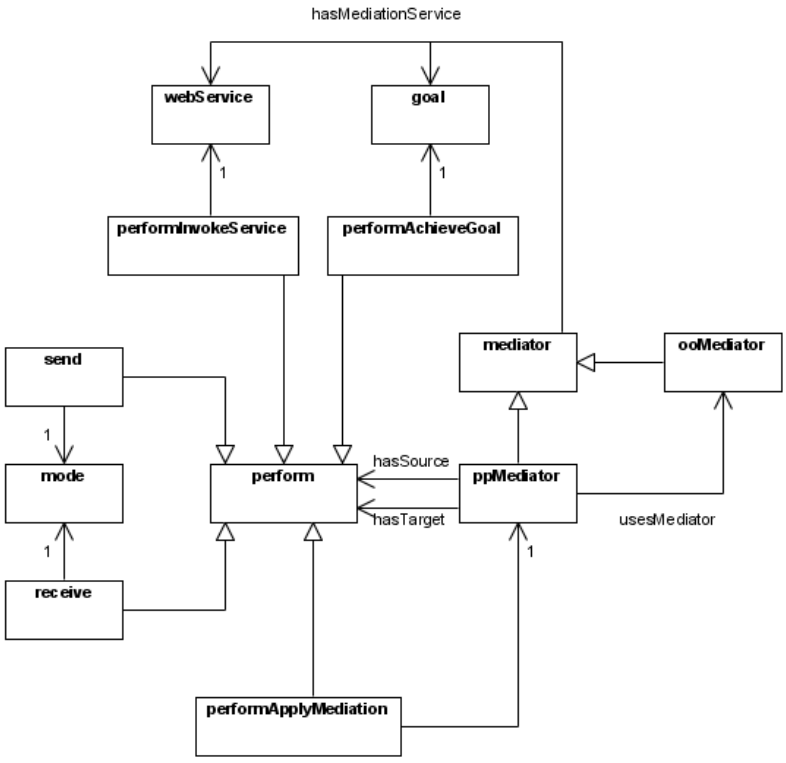


Fig. 4. Proposed Extension for WSMO Orchestration Meta-model

As described above for wg-mediators, pp-mediators have the ability to specify an oo-mediator and a mediation goal or service. In order to specify dataflow, the oo-mediator will identify which ontologies used by the source and the target are to be used, and a mediation goal will identify using its choreography which concepts from the orchestration state signature are to be taken from and written.

4 Cashew

The meta-model for workflow descriptions in Cashew is shown in Figure 5. The top half of the diagram shows the workflow operators which are subconcepts of *'cashewWorkflow'*. Each composes some collection of performs, shown centrally, which is extended to allow performance of workflows, hence hierarchical definition in the style of OWL-S. The other extension from the metamodel presented in the last section is the addition of a second novel mediator, pf-mediator. This allows the dataflow connection of performs to their containing workflow, avoiding the use of the specially-treated variable, with no ontological distinction, *'theParentPerform'* in OWL-S.

The workflow operators are divided into three types: *'Sequential'* depends on an ordered list of performances; *'Concurrent'* — called *'Split-Join'* in OWL-S — and *'Interleaved'* — called *'Any-Order'* in OWL-S, and renamed as a shortened form of the workflow pattern *'Interleaved Parallel Routing'* — both rely on an *unordered* set of performances; *choices* abstract over once-off choices and loops.

The subset of operators to which 'Internal Choice' is a superconcept represent exactly those in OWL-S — with *'If-Then-Else'* renamed after the workflow pattern *'XOR'* — but substituting the WMSO concept *'logicalExpression'* for the condition that the engine will evaluate to resolve the choice. In the case of *'XOR'* the condition will only be evaluated once to chose between the left and right performance; in the case of *'While'* and *'Until'*, after the left performance is evaluated, which will happen without evaluating the condition in the first instance with *'Until'*, the condition will be evaluated again.

In our previous semantics for OWL-S [20], we paid careful attention to the *'Any-Order'* operator, elided in other semantics [5]. In the informal semantics published in the specification [11] it is stated that only one performance at a time will be executed, and that the performance to be executed at run-time will depend on availability of input data, since component performances may communicate to supply one another with data. This data-driven characteristic is in contrast to the control-driven workflows in some other formalisms, such as *'flow'* in BPEL [12], which is due to WSFL [18].

In the spirit of this data-driven approach, since this happens to coincide with our own previous work [22], we offered an alternative semantics for *'Choose-One'*, where a non-deterministic choice would be made only between the 'ready' branches, *i.e.* those whose input has been provided. In the case that all branches are ready, this is an equivalent non-deterministic choice. In the case that different outputs can be produced, *e.g.* by the invocation of an operation — not considered in OWL-S, but expected in WSMO — this allows the choice to be resolved externally between subsequent performances depending on the different messages. Furthermore, given the extension to explicit message receipts from the client, having extended the types of performance, this becomes the 'classical' *'Deferred choice'* workflow pattern, which we therefore claim to generalise on, and name our operator after. We extend this 'external choice' to be able to decide also loops, in *'Deferred While'* and *'Deferred Until'*, where the loop is broken by the readiness of an unless/until branch.

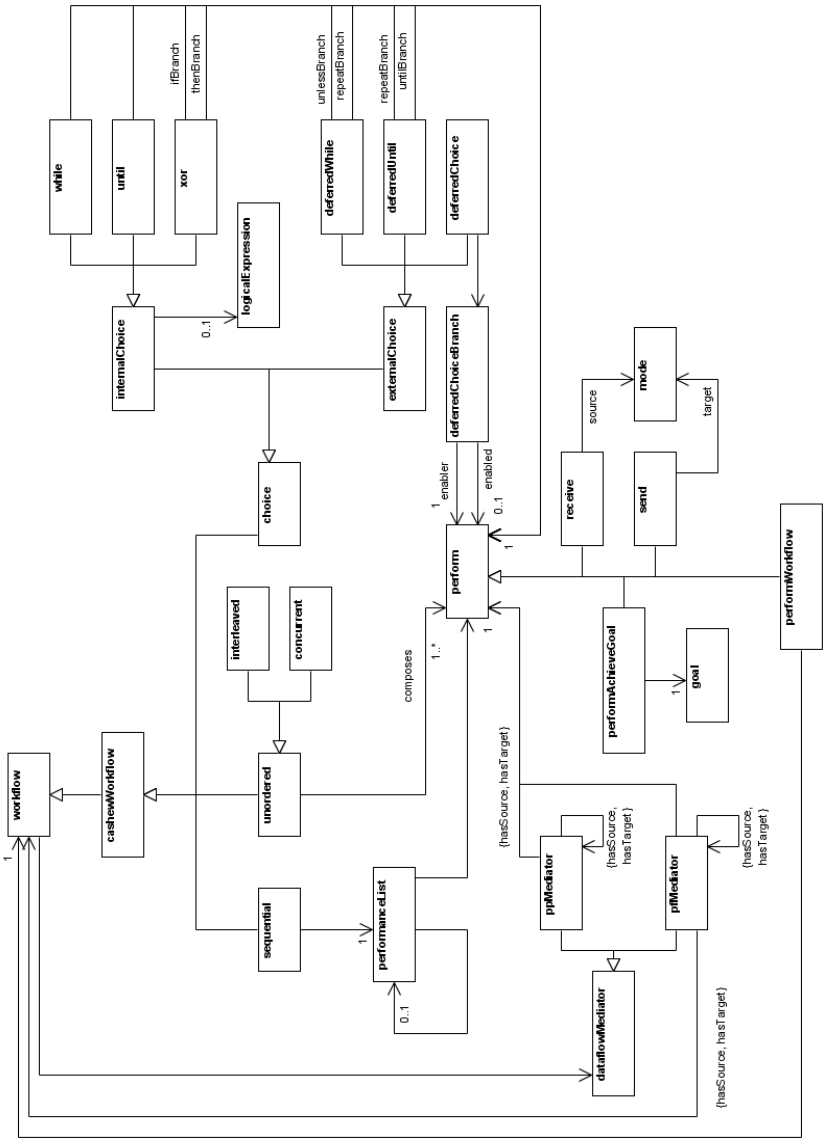


Fig. 5. Cashew Meta-model

5 Activity Diagrams

We now present the abstract model of a subset of UML 2 activity diagrams [14] as a UML class diagram together with Z constraints allowing to specify well-formed choreographies and orchestrations. In order to produce an unquestionable specification, we have chosen not to use the UML constraint language OCL, but instead a fragment of the Z language. This has several advantages:

- the limitations brought by the exclusive use of the dotted notation in OCL are overcome using Z, a language with extremely rich expressiveness
- all workflow well formedness rules can be presented unambiguously
- Z is extensible: it allows the declaration of user defined operators that complement the syntax. We use this feature to introduce the largely accepted dotted notation. As often as required and possible, OCL like dotted statements will be used
- Z set theoretic axioms translate naturally to set variables and constraints, used in the context of automatic constraint based composition

For brevity we do not display in the sequel the Z definitions of classes or relations as they strictly match the class diagrams. The reader can refer to [16] for a presentation of how Z can be used to specify constrained object models.
We begin with a short overview of activity diagrams, then we present the syntax, semantics and constraints of the constructs allowed in our subset.

5.1 Overview of UML2AD Diagrams

Diagrams An UML2AD diagram is a graph. Vertices are called nodes and may be internal actions, message in/out actions, workflow nodes (fork/join etc.) or also object nodes that represent the types of messages. The edges are directed and connect nodes to represent the flow of control or data between them.

More precisely, data flow edges are attached to their source or target actions via "pins". These pins specify the types of the tokens that may be produced or consumed by the action. They also have additional operational semantics that we ignore here: they occur as placeholders with limited capacity for tokens waiting to be consumed according to the "traverse to completion semantics". The full range of options that are available in the general AD language lead to utterly complex issues that we cannot cover here.

An alternative way of describing the types of messages in diagrams is through the use of "object nodes" in the middle of an edge. Of course, the types of tokens exchanged via a data flow edge must be compatible with the types declared for pins at both ends. A detailed description of UML2AD diagrams can be found in [14].

Token Flow AD edges describe the paths that may be followed by tokens, hence the "token flow". Tokens are produced, consumed, modified by actions. A specific node produces the initial (control) token required to initiate an activity

(the "FlowStart" node). Other nodes delete tokens: "FlowFinal" is a sink that deletes any incoming token without further side effect, whereas "ActivityFinal" deletes its incoming token as well as all currently active tokens in the diagram, hence resulting in stopping it.

Actions synchronize on all their input pins (they may execute only when a token is available on one corresponding output pin for all of them). After execution, actions present tokens with the proper types on all their output pins.

The main difference between the semantics induced by token flows compared to those of state machines stems from the fact that the number of tokens within an activity diagram may be arbitrarily high. The "state" of an activity graph hence remains implicit, whereas it is explicit in state machines.

The sequel presents the language of activity diagrams that we use for specifying / composing web services.

5.2 Activity Groups

Activity groups are sub-diagram containers. They are mostly needed to implement "interruptible" regions, which allow the deletion of all their tokens upon specific events.

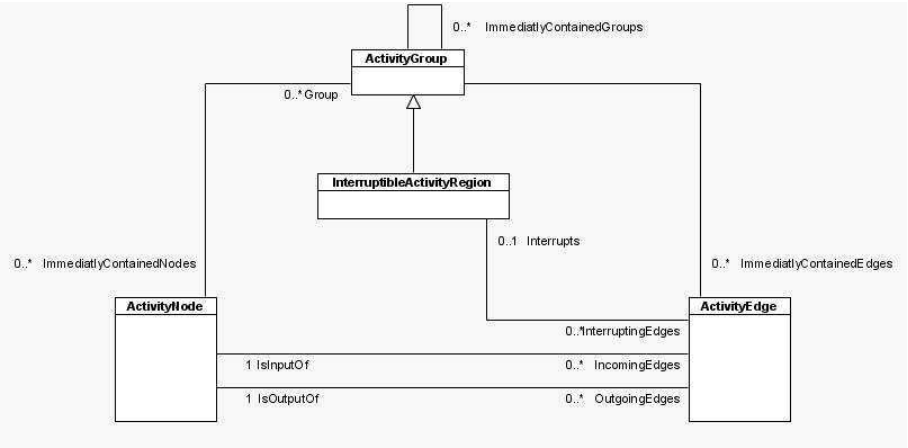


Fig. 6. Overview - Activity Groups

Semantics

- Groups: no special semantic, it just enables to group together a part of the activity. Web services can be represented as a group.
- InterruptibleRegions: used to model external choices. Whenever a token traverses an interrupting edge, all other tokens of the region are consumed.

Relations and roles

$$
\begin{array}{l}
interrupts : ActivityEdge \quad InterruptibleActivityRegion \\
isInputOf : ActivityEdge \quad ActivityNode \\
isOutputOf : ActivityEdge \quad ActivityNode \\
incomingEdges : ActivityNode \quad ActivityEdge \\
outgoingEdges : ActivityNode \quad ActivityEdge \\
immediatelyContainedGroups : ActivityGroup \quad ActivityGroup \\
immediatelyContainedNodes : ActivityGroup \quad ActivityNode \\
nodeGroup : ActivityNode \quad ActivityGroup \\
\hline
\forall\, n : ActivityNode;\ g : ActivityGroup \bullet \\
\quad g = nodeGroup(n) \Leftrightarrow n \in immediatelyContainedNodes(g) \\
\forall\, n : ActivityNode \bullet \\
\quad incomingEdges(n) = \{e : ActivityEdge \mid e.isInputOf = n\} \\
\forall\, n : ActivityNode \bullet \\
\quad outgoingEdges(n) = \{e : ActivityEdge \mid e.isOutputOf = n\}
\end{array}
$$

Constraints

– InterruptibleRegions: Interrupting edges have source in the region and target outside the region

$$
\begin{array}{l}
\forall\, x : ActivityEdge;\ y : InterruptibleActivityRegion \mid y = x.interrupts \bullet \\
\quad x.isOutputOf.nodeGroup = y \wedge \\
\quad x.isInputOf.nodeGroup \neq y
\end{array}
$$

5.3 Activity Nodes and Edges

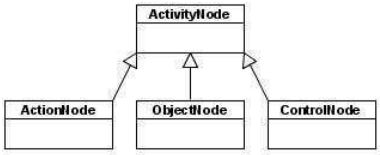


Fig. 7. Activity Nodes

Semantics The operational semantics of object and control flows are described in the UML as "traverse-to-completion" semantics. The aim of these semantics is to allow workflow not to enter undue self blocking states, that could be caused for instance by tokens mistakenly sent to an alternative outgoing path, and thus missing for a synchronization to occur via an other outgoing path. The

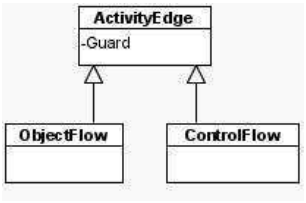


Fig. 8. Activity Edges

currently presented subset of UML2AD diagrams overcomes most difficulties by disallowing random alternative routes outgoing actions. In other words, when a token is produced by an action, it is presented to an output pin that has no more than one edge connected.

- Object Flows: carry data tokens
- Control Flows: carry control tokens.
- Guards: conditions expressing which decision node's outgoing edge will receive a token.

Attributes We define *Guard* as an uninterpreted set

$$[Guard]$$

and *else* a particular member of *Guard*:

$$else : Guard$$

We now specify the *guard* attribute as a partial function from *ActivityEdge* to *Guard*:

$$guard : ActivityEdge \quad Guard$$

Constraints

- ActivityEdge:
  - Only edges outgoing from a decision node can have a guard. Decision nodes are visually and formally presented with the other control nodes later in the document in Figure 10

$$\forall e : ActivityEdge; g : Guard \mid g = guard(e) \bullet e.isOutputOf \in DecisionNode$$

  - Only one edge outgoing from the same decision node can have an else condition as the guard.

$$\forall n : DecisionNode \bullet \\ \#\{e : ActivityEdge \mid n = isOutputOf(e) \land else = guard(e)\} = 1$$

– Control Flow:
Control flows may not have object nodes at either end

$$\forall\, e : ControlFlow \bullet \\ e.isInputOf \notin ObjectNode \wedge e.isOutputOf \notin ObjectNode$$

5.4 Action and Object Nodes

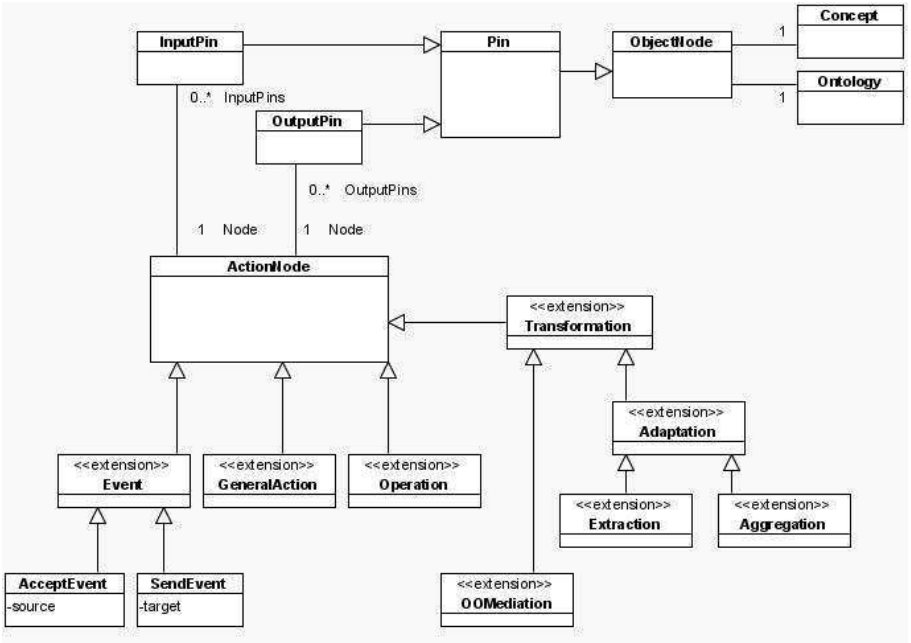


Fig. 9. Action Nodes and Object Nodes

Semantics

– ActionNode :
- Denotes that a local action is realized at this node
- Pins are used to receive and send data tokens
- The inputs are synchronized (all incoming edges and input pins have to carry a token for the action to start)
– OOMediation: This is an additional construct from UML2AD specification
– AbstractEvent: this is an additional construct from UML2AD specification. Not executable, i.e any AbstractEvent has to be specialized. source and target only applies to specify the corresponding sender of receiver in an orchestration.

- Operation: this is an additional construct from UML2AD specification. Specifies a local action as being an operation
- Adaptation: this is an additional construct from UML2AD specification. Specifies an aggregation or extraction of message(s)

Relations

$$
\begin{array}{l}
concept : ObjectNode \quad Concept \\
ontology : Concept \quad Ontology \\
node : Pin \quad ActionNode \\
inputPins : ActionNode \quad InputPin \\
outputPins : ActionNode \quad OutputPin \\
\hline
\forall\, y : ActionNode \bullet inputPins(y) \cap outputPins(y) = \\
\forall\, x : Pin;\ y : ActionNode \bullet node(x) = y \Leftrightarrow x \in inputPins(y) \cup outputPins(y)
\end{array}
$$

Constraints

- ObjectFlow:
  - Object Flow connects exclusively object nodes, decision nodes, merge nodes, fork nodes and join nodes.

$$
\begin{array}{l}
\forall\, f : ObjectFlow \bullet \\
\quad \{f.isInputOf\} \cap ObjectNode \neq \\
\quad \vee\ \{f.isInputOf\} \cap DecisionNode \neq \\
\quad \vee\ \{f.isInputOf\} \cap MergeNode \neq \\
\quad \vee\ \{f.isInputOf\} \cap ForkNode \neq \\
\quad \vee\ \{f.isInputOf\} \cap JoinNode \neq \\
\forall\, f : ObjectFlow \bullet \\
\quad \{f.isOutputOf\} \cap ObjectNode \neq \\
\quad \vee\ \{f.isOutputOf\} \cap DecisionNode \neq \\
\quad \vee\ \{f.isOutputOf\} \cap MergeNode \neq \\
\quad \vee\ \{f.isOutputOf\} \cap ForkNode \neq \\
\quad \vee\ \{f.isOutputOf\} \cap JoinNode \neq
\end{array}
$$

  - The downstream object node type must be the same of the upstream object node type

$$
\begin{array}{l}
\forall\, f : ObjectFlow;\ s, t : Pin \mid \\
\quad s = isOutputOf(f) \wedge t = isInputOf(f) \bullet \\
\quad\quad s.ontology = t.ontology \\
\forall\, f : ObjectFlow;\ s, t : Pin \mid \\
\quad s = isOutputOf(f) \wedge t = isInputOf(f) \bullet \\
\quad\quad s.concept = t.concept
\end{array}
$$

– AcceptEvent:
No incoming activity edge

$$\forall\, e : ActivityEdge \bullet e.isInputOf \notin AcceptEvent$$

– SendEvent :
No outgoing activity edge

$$\forall\, e : ActivityEdge \bullet e.isOutputOf \notin SendEvent$$

5.5 Control Nodes

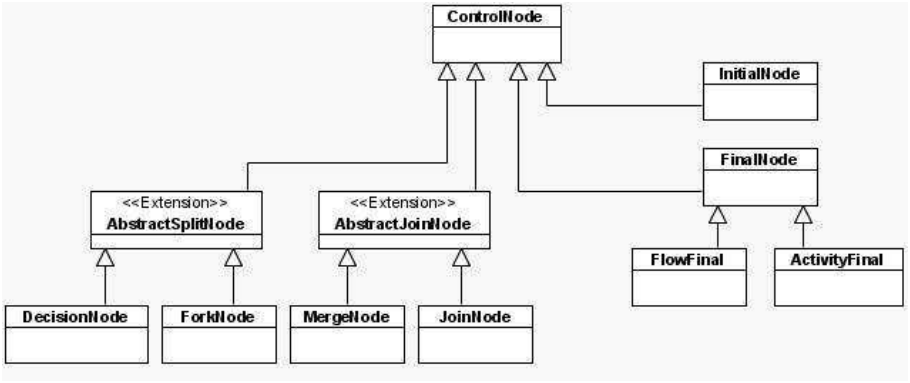


Fig. 10. Control Nodes

Semantics

– AbstractSplit: this is an additional construct from UML2AD specification. Not executable : any AbstractSplit has to be specialized
– AbstractJoin: this is an additional construct from UML2AD specification. Not executable : any AbstractJoin has to be specialized
– MergeNode: any token offered on any incoming edge is offered to the outgoing edge
– DecisionNode: each token arriving can traverse to only one outgoing edge
– ForkNode: incoming token duplicated to outgoing edges
– JoinNode: when all incoming edges have tokens, one is created on outgoing edge. Only one incoming edge can be an object flow. Outgoing edge can be an object flow only if there is an object flow among the incoming edges (in this case, the incoming data token is sent to the outgoing edge)
– Flow Final: consumes one token
– Activity Final: all tokens in the activity are consumed

Constraints

- AbstractSplit:
  1 incoming edge only

$$\forall\, x : \mathit{AbstractSplit} \bullet \#(x.\mathit{incomingEdges}) = 1$$

- AbstractJoin:
  1 outgoing edge only

$$\forall\, x : \mathit{AbstractJoin} \bullet \#(x.\mathit{outgoingEdges}) = 1$$

- JoinNode:
  Only one incoming edge is an object flow

$$\forall\, x : \mathit{JoinNode} \bullet \#((x.\mathit{incomingEdges}) \cap \mathit{ObjectFlow}) \leq 1$$

- InitialNode:
  no incoming edge

$$\forall\, x : \mathit{InitialNode} \bullet x.\mathit{incomingEdges} =$$

- FinalNode:
  no outgoing edge

$$\forall\, x : \mathit{FinalNode} \bullet x.\mathit{outgoingEdges} =$$

- DecisionNode: the edges coming into and out of a decision node must be either all object flows or all control flows
- MergeNode: the edges coming into and out of a decision node must be either all object flows or all control flows

$$\begin{aligned}&\forall\, x : \mathit{ActivityNode} \mid x \in \mathit{DecisionNode} \cup \mathit{MergeNode} \bullet \\ &\quad (x.\mathit{incomingEdges} \cup x.\mathit{outgoingEdges}) \subset \mathit{ObjectFlow} \vee \\ &\quad (x.\mathit{incomingEdges} \cup x.\mathit{outgoingEdges}) \subset \mathit{ControlFlow}\end{aligned}$$

6 Representing Cashew in UML

The alliance with workflow patterns allows a standard mapping from most parts of Cashew directly into UML Activity Diagrams [30]. Still, the transformation presented here introduces some modifications in order to favour workflow composability as well as it makes the appropriate links to the WSMO meta-model. The reader is referred to Figure 5 for a complete list of the workflow operators provided by Cashew.

The **Sequential** operator has a direct and obvious transformation into UML Activity Diagrams by connecting sequenced performances with control flows. This operator does not present any difficulties with respect to its composability since it provides by definition two unique nexus, i.e. the first and the last performances, where other workflows can be connected.

Figure 11 illustrates how to represent the **Concurrent** operator in UML Activity Diagrams. This is in fact composed of a UML **AND-split** followed by an **AND-join** meaning that every thread must complete. The performances in between representing the different activities that have to be performed concurrently. Doing so paves the way for composing workflows by providing a central point where preceding and subsequent workflows can be connected as illustrated in the figure by the top and bottom grey activity boxes respectively.

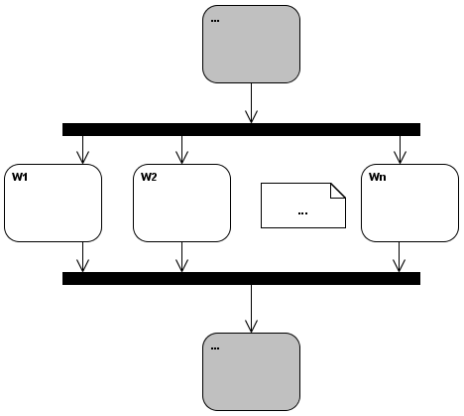


Fig. 11. Concurrent pattern in UML

Interleaved workflows are transformed in a similar fashion to that shown in [30] where signals are used as semaphores, see Figure 12 for an example with three interleaved performances **V**, **W** and **X**. Before a performance can start, a signal has to be received. In UML Activity Diagrams if several receivers are ready to consume an event, only one action accepts the event. Therefore, once a performance is completed, a signal needs to be sent again so that other performances can be executed. As a consequence no order is established between the performances a priori, which leads to an arbitrary order of execution as dictated by the Interleaved Parallel Routing workflow pattern [28]. Finally, in order to support composability of workflows, signals are merged.

A **XOR** workflow between performances **V** and **W** is depicted in Figure 13. A decision node, based on the condition specified by the axiom **a**, connects both performances and a final merge node ensures the final result is composable.

The **While** and **Until** constructs, depicted in Figures 14 and 15 respectively, are both very similar to the XOR-type workflow. The main difference is that they do not require a merge node as a central point for connecting to subsequent workflows, since there is only one outgoing path. This is illustrated by means of the grey box connected to the 'else' branch. Both constructs differ in that the performance **V** is always executed at least once in the Until pattern, whereas it might never be executed in While-type workflows. In fact as shown in the

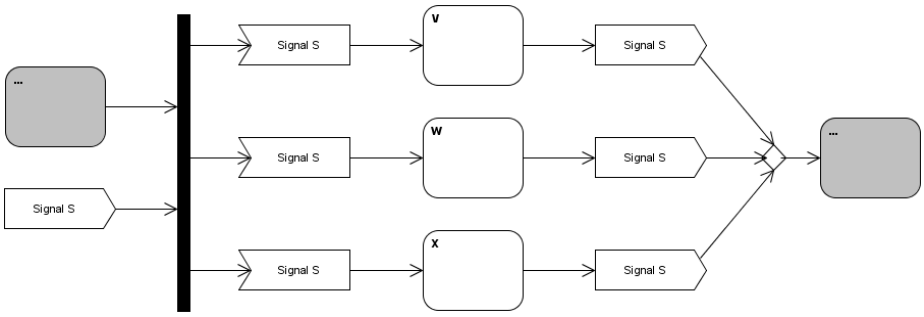


Fig. 12. Interleave pattern in UML

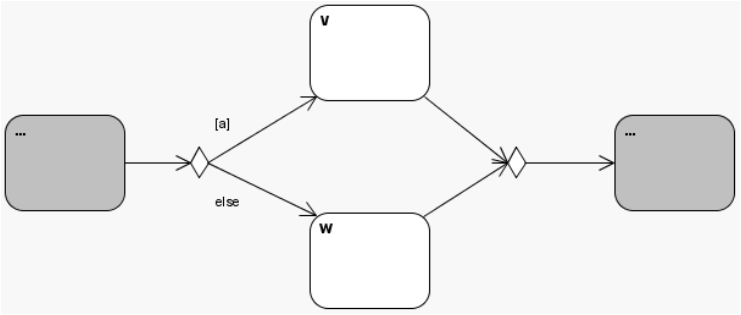


Fig. 13. XOR pattern in UML

figures, the preceding performance connects to the decision node in the case of the While construct as opposed to Until-type workflows where it directly links to the performance **V**.

The **Deferred Choice** pattern is shown in Figure 16. Deferred Choice starts concurrently as the control node is split between various Atomic Performances. Atomic Performances can be either a Perform Goal, a Perform Send or a Perform Receive. These Atomic Performances will determine at runtime the branch to be executed. Once a branch is selected, the subsequent performance will be executed preempting the other possible branches. To support this, Atomic Performances are within an **interruptible region** and interrupting edges connect them to their respective subsequent performances, i.e. performances **V** and **X** in Figure 16. It is worth noting that these performances are optional, e.g. the action to be performed could simply be sending a message. Finally, for the sake of composability all performances connect to a 'merge node' which represents the unique nexus where following performances are to be connected.

In a similar fashion to the Deferred Choice pattern, **Deferred While** (see Figure 17) and **Deferred Until** (see Figure 18) are based on the use of interruptible activity regions. It is however worth noting that although the Deferred Choice pattern can include as many branches as desired, both Deferred While

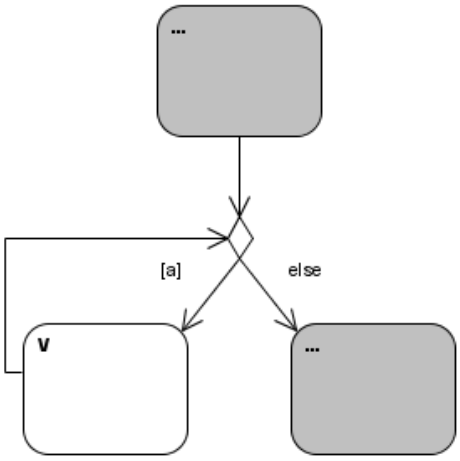


Fig. 14. While pattern in UML

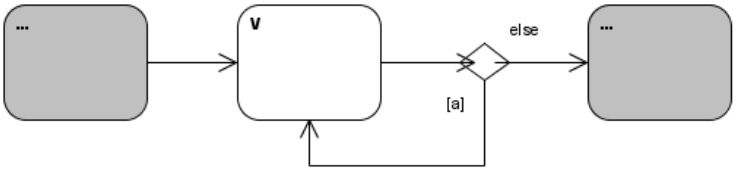
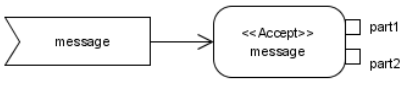


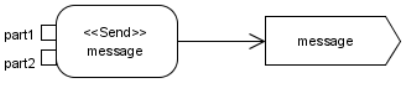
Fig. 15. Until pattern in UML

and Deferred Until only include two branches, one being the iteration body and the other representing the concluding Atomic Performance. Again, the difference between Deferred While and Deferred Until lies in that the iteration branch, i.e. the Performance **V**, is executed at least once in the Deferred Until pattern. Therefore, and to cater for composability, the Deferred While pattern includes a merge node where both the iterating performance and preceding workflows connect to. Conversely, in the Deferred Choice the incoming control flow is directly connected to the Performance **V** ensuring that it is at least executed once. Finally both patterns include a concluding branch with an Atomic Performance which also represents the outgoing nexus.

Finally, in order to define the dataflow each send and receive performance is associated with a UML *action* with *pins* that represent the atomic parts of the message, see Figures 16(a) and 16(b). In this way dataflow can be represented by connecting pins together with UML *object flow edges*.



(a) Receive in UML



(b) Send in UML

7 Example

As an example of the use of the three-level model, we give an overview of the example from [23], based on a telecoms industry use case. This concerns a composite business process to order three related products as a 'bundle'; a modem, the network connection and a PC. These orders are checked in a strict order due to the likelihood of failure, starting with the network, where many consumer lines may be incapable of supporting certain types of connection — in particular DSL-based ones.

Each of the three services to check the respective product has a choreography as diagrammed in Figure 19(a), and each service to confirm the order has a choreography as diagrammed in Figure 19(b). We note that while each has a receive event followed by some response event, the ModemRequest service has an internal (XOR) choice between a subsequent failure or success event.

The representation of this choreography is shown at each of the three-levels in Figure 20. We note that the Activity Diagrams version directly encodes these representations, that the Cashew workflow directly represents the receive sequentially followed by this xor-choice between two sends, and that the ASM uses all three recursive transition rule types and direct manipulations on the state signature (since this is a choreography). These three representations are shown in an extension of the standard DIP tool, WSMO Studio [7], supporting the extended WSML grammar via an extension to the standard object model for WSMO/L, WSMO4J[8].

A section of the orchestration between the six services in shown as an Activity Diagram in Figure 21. We note that the choreography's internal choice has been turned into a deferred choice in the orchestration since, as a client to these services, the orchestration engine does not resolve this choice directly, but waits for the response message in order to choose which branch to follow.

A full treatment of this example, including an orchestration on all three levels, may be found in [23]; space prevents us from reproducing this here.

8 Conclusions and Further Work

In this paper we have shown a three-level framework for the ontological representation of behavioural models and how this is compatible with WSMO. We have shown how Cashew and the UML can be fitted into this model, allowing a relationship to be established with existing communities and their tools. Most importantly, we have sketched, at the UML level, how a standard Workflow Patterns-based translation to different representations is encoded in Cashew and can be employed to create ASM-based models for compatibility with tools implementing the current WSMO/L standards, like WSMX.

[7] http://www.wsmostudio.org
[8] http://wsmo4j.sourceforge.net

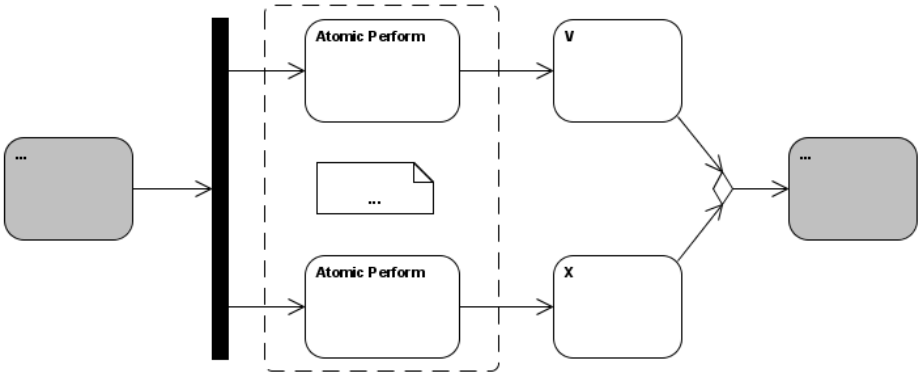


Fig. 16. Deferred Choice pattern in UML

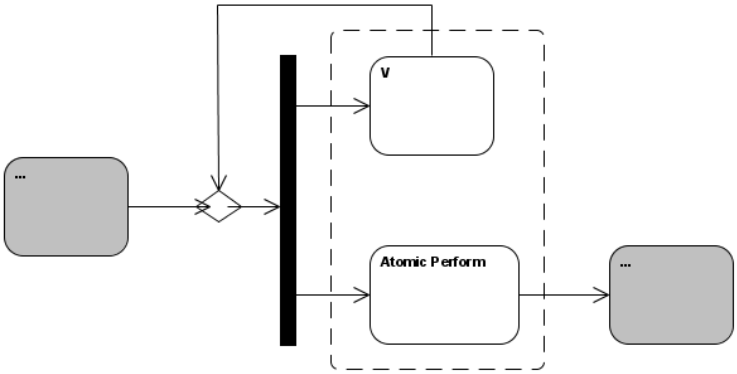


Fig. 17. Deferred While pattern in UML

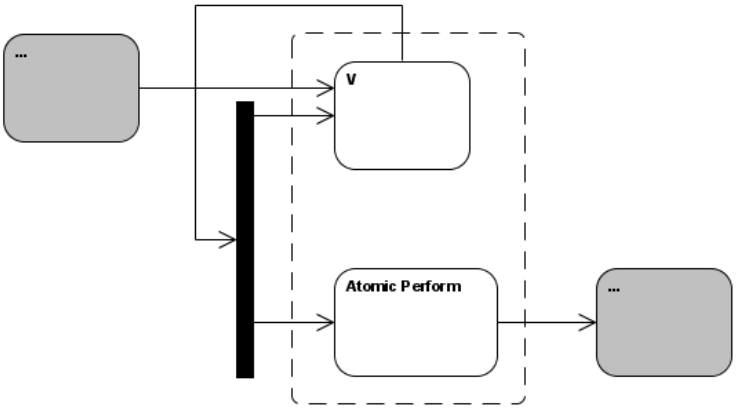
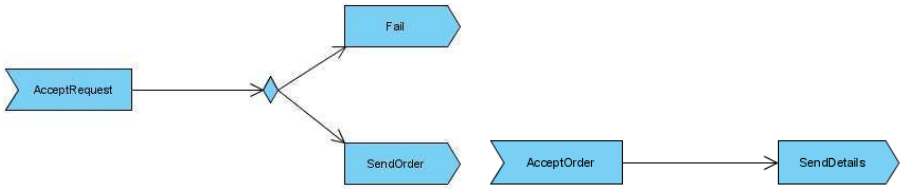


Fig. 18. Deferred Until pattern in UML



(a) Modem Request Choreography (b) Modem Confirm Choreography

Fig. 19. Example Choreographies as Activity Diagrams

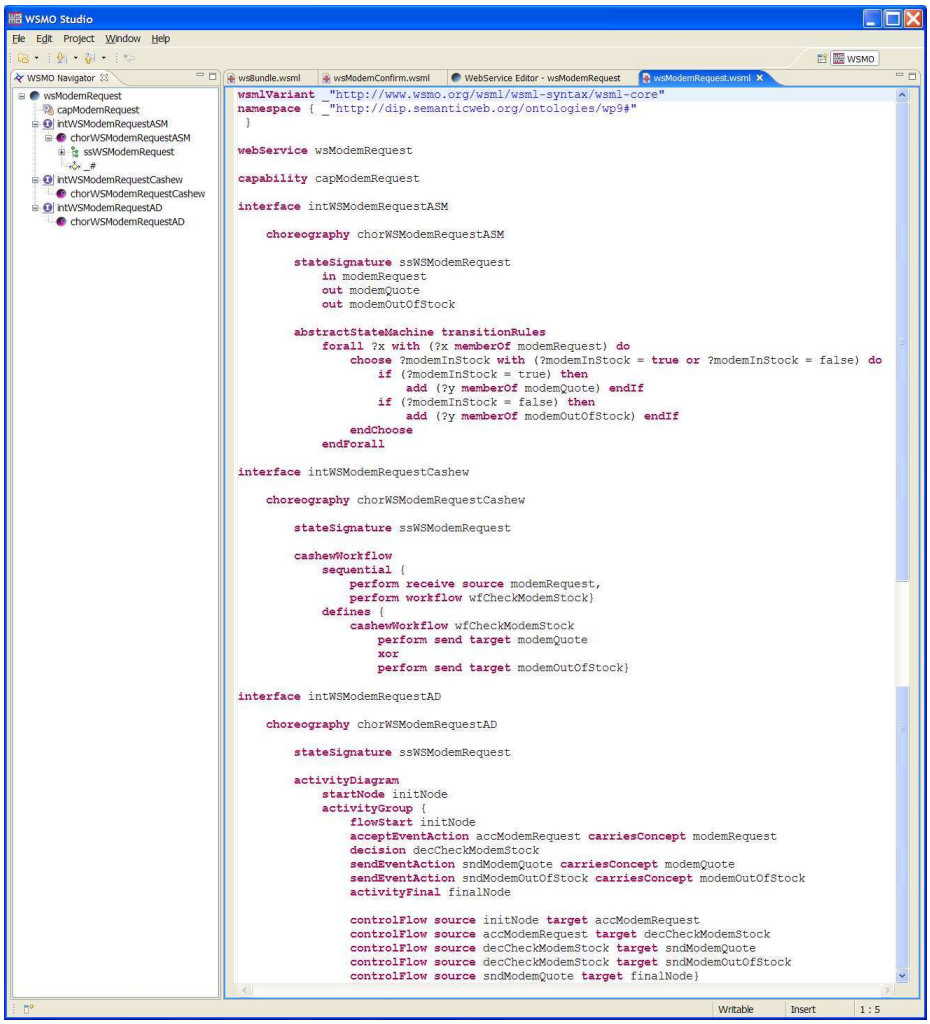


Fig. 20. Example Choreography on 3 levels in WSML

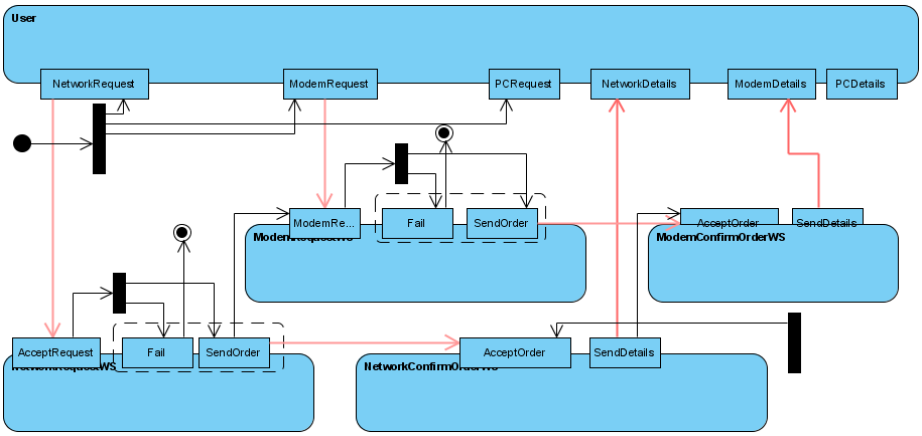


Fig. 21. Example Orchestration as an Activity Diagram

Existing implementations in the Cashew scope involve an interpreter for Cashew in the IRS-III [6], and an on-going implementation of translation from Cashew to UML and a (partial) reverse-translation, and to ASMs. At the ASM layer, WSMX [15], the open-source reference implementation for WSMO, has implemented the extended ASM grammar presented here. A prototype composer in the DIP project uses UML2AD descriptions of WSMO web service choreographies for automatic composition, and exports its results as valid UML2AD orchestrations. The result, when it can be translated to Cashew, can then be readily executed on the IRSIII platform. The proposed three layer architecture has hence proved its functional validity on several industrial use cases. The composer also involves a composition goal language that cannot be presented here, but is specified in [1, 2]. Finally, as discussed above, both WSMO4J and WSMO Studio have alternatives that support the extended grammars used in DIP, and the meta-model proposed, as discussed in Section 3, has been proposed to the WSMO Working Group for potential standardisation [21].

We also have reason to claim the general utility of our model due to recent work in the SUPER project[9], which concerns semantics representation and extension of business process modelling (BPM) [17]. In this application, recalling Figure 1, both EPCs [27] and BPMN [24] are used in the diagram layer, and semantically-extended version of BPEL4WS [12], called BPEL4SWS, is used at the Execution Layer. In between we are developing an ontology called BPMO, Business Process Modelling Ontology, based on Workflow Patterns like Cashew, but extended with more business-oriented tasks, which will allow translation between the two, based on an extension of [25]. Finally the approach suggested in Section 2.3 will be applied to BPMO, *i.e.* the development of ontological behavioural reasoning via an axiomatisation of process-algebraic semantics. These results will first be made available via deliverables at the SUPER website.

[9] http://www.ip-super.org

References

1. Patrick Albert, Christian de Sainte Marie, Mathias Kleiner, and Laurent Henocque. Specification of the composition prototype. http://dip.semanticweb.org/deliverables.html, December 2005. DIP Deliverable D4.12.
2. Patrick Albert, Christian de Sainte Marie, Mathias Kleiner, and Laurent Henocque. Composition prototype. http://dip.semanticweb.org/deliverables.html, June 2006. DIP Deliverable D4.15.
3. Patrick Albert, Laurent Henocque, and Mathias Kleiner. Configuration-based workflow composition. In *in proceedings of IEEE International Conference on Web Services (ICWS'05)*, pages 285–292. IEEE, 2005.
4. Patrick Albert, Laurent Henocque, and Mathias Kleiner. A constrained object model for configuration based workflow composition. In Bussler C. et Al, editor, *Revised and Selected Papers, Business Process Management - BPM 2005 Workshops*, volume 3812 of *Lecture Notes in Computer Science*, pages 102–115. Springer, janvier 2006.
5. A. Ankolekar, F. Huch, and K. Sycara. Concurrent semantics for the web services specification language DAML-S. In *Proc. 5th Intl. Conf. on Coordination*, volume 2315 of *LNCS*, 2002.
6. L. Cabral, J. Domingue, S. Galizia, A. Gugliotta, B. Norton, V. Tanasescu, and C. Pedrinaci. IRS-III – A Broker for Semantic Web Services based Applications. In *In Proc. of the 5th International Semantic Web Conference (ISWC 2006), Athens(GA), USA*, 2006.
7. M. Dumas and A. H. M. ter Hofstede. UML Activity Diagrams as a workflow specification language. In *Proc. 4th Intl. Conf. on the Unified Modeling Language (UML)*, number 2185 in LNCS, 2001.
8. D. Roman *et al.* Orchestration in WSMO (working version). http://www.wsmo.org/TR/d15/v0.1/, January 2005.
9. D. Roman *et al.* Web service modeling ontology WSMO v1.2. http://www.wsmo.org/TR/d2/v1.2/, April 2005.
10. D. Roman *et al.* Ontology-based choreography of wsmo services v0.3. http://www.wsmo.org/TR/d14/v0.3/, May 2006.
11. David Martin *et al.* OWL-S: Semantic markup for web services. http://www.daml.org/services/owl-s/1.1/overview/, 2004.
12. S. Thatte *et al.* Business process execution language for web services version 1.1. ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf, 2003.
13. D. Fensel, H. Lausen, A. Polleres, J. de Bruijn, M. Stollberg, D. Roman, and J. Domigue. *Enabling Semantic Web Services. The Web Service Modeling Ontology*. Springer, 2006.
14. Object Management Group. UML 1.4.2 specification. Technical Report ISO/IEC 19501, ISO, 2005.
15. A. Haller, E. Cimpian, A. Mocan, E. Oren, and C. Bussler. WSMX - a semantic service-oriented architecture. In *Proc. 4th Intl. Semantic Web Conference (ISWC 2005)*, number 3729 in LNCS, 2005.
16. Laurent Henocque. Modeling object oriented constraint programs in z. *RACSAM (Revista de la Real Academia De Ciencias serie A Mathematicas)*, special issue on Symbolic Computation in Logic and Artificial Intelligence(98 (1–2)):127–152, 2004.

17. M. Hepp, F. Leymann, J. Domingue, A. Wahler, and D. Fensel. Semantic business process management: A vision towards using semantic web services for business process management. In *Proceedings of the IEEE ICEBE 2005*, pages 535–540, 2005.
18. F. Leymann. Web services flow language (WSFL 1.0). http://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf, 2001.
19. B. Norton. Experiences with OWL-S, directions for service composition: The Cashew position. In *OWL: Experiences and Directions Workshop (co-located with ESWC 2005)*, 2005. http://www.mindswap.org/OWLWorkshop/sub23.pdf.
20. B. Norton, S. Foster, and A. Hughes. A compositional semantics for OWL-S. In *Proc. 2nd Intl. Workshop on Web Services and Formal Methods (WS-FM 05)*, number 3670 in LNCS, Sept 2005.
21. B. Norton and T. Haselwanter. Dataflow for orchestration in WSMO. http://www.wsmo.org/TR/d15/d15.1, July 2006.
22. B. Norton, G. Lüttgen, and M. Mendler. A compositional semantic theory for synchronous component-based design. In *14th Intl. Conference on Concurreny Theory (CONCUR '03)*, number 2761 in LNCS. Springer-Verlag, 2003.
23. Barry Norton, Carlos Pedrinaci, Jens Lemcke, Laurent Henocque, Mathias Kleiner, and Gabi Vulcu. Ontology for web services choreography and orchestration. http://dip.semanticweb.org/deliverables.html, January 2007. DIP Deliverable D3.9.
24. OMG. Business process modeling notation (BPMN) specification. Technical report, Object Management Group, 2006.
25. C. Ouyang, M. Dumas, A. ter Hofstede, and W. van der Aalst. From BPMN process models to BPEL web services. In *Proceedings of the IEEE International Conference on Web Services (ICWS'06)*.
26. M. Stollberg and D. Fensel. Ontology-based choreography of WSMO services. http://www.wsmo.org/TR/d14/, July 2005.
27. W. M. P. van der Aalst. Formalization and verification of event-driven process chains. *Information & Software Technology*, 41(10):636–650, 1999.
28. W. M. P. van der Aalst, A. H. M ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(3):5–51, June 2003.
29. P. Wohed, W. M. P. van der Aalst, M. Dumas, A. H. M. ter Hofstede, and N. Ruseell. Pattern-based analysis of UML activity diagrams. BETA Working Paper Series WP 129, Eindhoven University of Technology, 2005.
30. Petia Wohed, Wil M. P. van der Aalst, Marlon Dumas, Arthur H. M. ter Hofstede, and Nick Russell. Pattern-based analysis of the control-flow perspective of uml activity diagrams. In *Proceedings of Conceptual Modeling - ER 2005, 24th International Conference on Conceptual Modeling*, pages 63–78, 2005.

Acknowledgements

This work is supported by the DIP project, an Integrated Project (FP6 - 507483), and the SUPER project (FP6 - 026850), both from the European Union's IST program, and by the ILOG Company.

We would also like to particularly acknowledge the contribution of all the IRS-III team working under John Domingue at KMi.